CHALMERS
UNIVERSITY OF TECHNOLOGY

Adaptive Stream-based Shifting Bottleneck Detection in IoT-based Computing Architectures

Adaptive Stream-based Shifting Bottleneck Detection in IoT-based Computing Architectures

Hannaneh Najdataei, Mukund Subramaniyan, Vincenzo Gulisano, Anders Skoogh, Marina Papatriantafilou
Chalmers University of Technology, Sweden
{hannajd,mukunds,vincenzo.gulisano,anders.skoogh,ptrianta}@chalmers.se

***Abstract*—Cloud computing is revolutionizing the backbone of data analysis applications, including industrial ones. One of its main pillars is the separation of the logic with which data is accessed (e.g., to study the efficiency of a manufacturing system) from the actual hardware (e.g., server) that maintains and analyses the data. Large distributed cyber-physical systems enabled by, among other technologies, the Internet of Things (IoT), made nonetheless clear that "what to do" with the data and "where to do it" are not disjoint problems; i.e., cloud computing on its own is not enough. Fog and edge computing have emerged as complementary options, to distribute the analysis, helping with challenges by means of close-to-the-source data analysis.**

We show for a key problem for industrial processes, that of shifting bottleneck detection, how to take advantage of such multi-tier computing architectures, to perform continuous and configurable analysis of data from Manufacturing Execution Systems. We propose a processing framework, *STRATUM*, and an algorithm, *AMBLE*, for continuous, data stream processing. *STRATUM* seamlessly distributes and parallelizes the processing across the tiers and *AMBLE* guarantees consistent analysis in spite of timing fluctuations, which are commonly introduced due to e.g. the communication system; it also achieves efficiency through appropriate data structures for in-memory processing. The experimental study on a real-world dataset, taken from a production line over two years and including 8.5 million entries, shows the benefits of the proposed solution in enabling configurable and efficient analysis.

I. Introduction

A. Motivation

Throughput is an important indicator to measure production system performance [6]. It is often affected by arbitrary disruptions in the machines, such as fluctuations in the cycle time, down times and minor stops [11]. Machines in the production system can thus form *throughput bottlenecks* [16]. Improvements such as cycle time reductions and dynamic buffering are prioritizing the *bottleneck machines* for activities to improve the throughput [7], [15], [20].

Regarding bottleneck detection, methods in the extensive literature about this challenging problem, use information about e.g., *blockage, turning-point, inter-departure-time-variance, shifting-bottleneck, active-period-percentage* (cf. [3] and references therein). Detecting the bottlenecks *efficiently* can have large impact in productions systems. Even more important is to detect them at *configurable time granularity* (e.g. the bottlenecks of minutes, hours, shifts, days, months, etc), since the latter can provide key indicators to the system analysts for improvements in the production pipelines and even support adaptive, dynamic pipelines. Further, given the new possibilities with high-rate sensors, it is useful to utilize the continuous flows of data to understand properties about the system in a *continuous fashion*, to process them on-the-fly and to understand how the results can vary with time.

To define methods for continuous data analysis, the *data stream processing* paradigm is complementing the conventional *store-then-process* one [2]. The former processes continuous flows of data on-the-fly, while the latter first stores the data and then runs the analysis on it. The widely adoption of stream processing is motivated by the increase in data generated at high rates by sensors in various application domains, since the data streaming paradigm offers solutions for efficient data analysis, overcoming limitations of other methods in latency and system memory [18]. *Stream Processing Engines* (SPEs) are software systems designed to provide high-level programming interfaces for stream processing. Stateful analysis in stream processing is commonly done in defined *windows* over the streams, to capture the analysis relevant time-horizon.

The design of algorithmic approaches that can enable stream processing is problem-dependent and hence appropriate problem-analysis methods are required. Moreover, enabling configurable ways for utilizing the computing architectures in cyber-physical, IoT-based systems in the cloud, fog, and edge layers, shown in Figure 1, where the distribution of data processing is enabled along the *Things-to-Cloud* continuum [1], is another issue to be addressed.

B. Challenges and Related Work

Regarding bottleneck detection, the method introduced in [17], forms a prominent approach that has the potential of enabling detection of bottlenecks in continuous fashion, by defining *momentary and shifting bottlenecks*. The method has been originally proposed based on a discrete event model (DES) of a simulation system, which was an interesting

The authors would like to thank the FFI programme (funded by VINNOVA, the Swedish Energy Agency, and the Swedish Transport Administration) for their funding of the Data Analytics in Maintenance Planning research project (DAiMP) [Grant number: 2015-06887] and the Swedish Foundation for Strategic Research (SSF), for their funding of project FiC [Grant number: GMT14-0032], under which this research was conducted. The authors would also like to thank Anders Ramstrm, who furnished the real-time data from a real-world production system.

978-1-7281-0303-7/19/$31.00 ©2019 IEEE

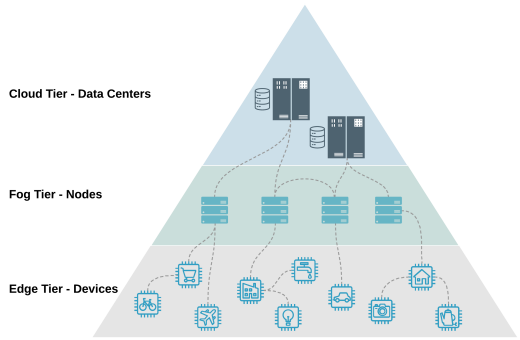


Fig. 1: *Multi-tier data processing architecture including cloud, with computing devices ranging from single-board, resource constrained ones to high-end servers.*

platform to demonstrate it. However, it was identified that the method needs further development to suit the real-world industrial shop floor practice of bottleneck management; towards that, a data-driven approach for the shifting bottleneck identification was proposed in [19], using the online data collected from Manufacturing Execution Systems (MES), to facilitate it for the shop-floor engineers to understand the production system behavior in real-time and thus enable faster actions on the bottlenecks. Although the latter has been an important contribution towards improving industrial practice, the method needs to be further developed to yield results continuously and allow multiple, configurable time granularity, that can be enabled in terms of *time-windows* in the stream processing terminology. Moreover, a critical question in the data-driven shifting algorithm application is the speed at which the algorithm detects the real-time bottlenecks and yield insights (i.e. elapsed time from data generation process to detection of bottlenecks). Existing literature does not include an analysis or an approach that pipelines data generation and processing. Given that data rates are increasingly high, the existing methods can be prohibitive for very high volumes of data, that are becoming the norm in evolving systems. Hence, efficient stream processing is a challenging necessity from this point of view as well.

C. Goals and Contributions

We aim at an efficient streaming algorithm for the shifting bottleneck detection method of [17], [19] and a configurable methodology to enable deployment considering practicalities in actual systems, including multi-tier computing architectures of IoT-based systems. Our contributions provide:

- a 2-tier framework (*STRATUM*) to support configurable and automated analysis, leveraging stream processing and enhancing task parallelism by distributing the work on embedded processing units at the machines, as well as the analysis center; the lower tier is responsible for data validation and filtering, while the upper one takes care of the combined data-stream analysis; the system analyst provides information about the parameters, e.g. number of machines, working hours, size of the processing window;
- an efficient algorithm (*AMBLE*) for streaming shifting-bottleneck-detection, for fine-grained continuous processing, that also preserves determinism (implying consistent results despite timing variation in e.g. the communication network), thanks to leveraging recent research contributions, namely the ScaleGate [8] data object;
- complexity analysis of *AMBLE*, showing constant overhead per data point, depending only on the number of machines, which is fixed for each configuration;
- an explanation of how the methodology enables configurable analysis, through parallel instances of streaming operators with different window sizes at the upper tier;
- experimental evaluation of the methods, using over 8.5 million data points, collected from a production system over two years; the evaluation shows the effectiveness of *STRATUM* and *AMBLE* algorithm in providing multiple-configuration results in a timely fashion.

Given the challenges in the previous subsection, these results can influence significantly the state of the art in the area.

II. Data Stream Processing Background

In data streaming, each data source produces a stream, that carries data records, aka *tuples*, sharing a common schema $< ts, A_1, A_2, ..., A_n >$ where ts is the creation timestamp and $A_1, ..., A_n$ are the attributes of the tuple. Data stream processing applications are modeled as Directed Acyclic Graphs (DAGs) where vertices represent processing *operators* and edges specify how tuples flow from data sources through several processing operators and eventually, to the data sinks where results are delivered.

A processing operator in data streaming, is either *stateless*, i.e. does not keep any state, or *stateful*, i.e. maintaining a state that depends on previous tuples. Due to the unbounded nature of data streams, stateful operators compute over a *window* of the stream which contains the most recent tuples. In this paper, we use *tumbling windows* which are fixed size, non-overlapping time intervals. Tumbling windows are defined by *size* parameter, i.e. the length of the window, commonly in time-units. SPEs can provide various operators, but a subset of basic ones is common among all, including:

- *Filter*, a stateless operator that for each input tuple produces at most one output tuple by forwarding or discarding the former, based on a filtering condition.
- *Map*, a stateless operator that transforms each tuple from one schema to another and produces zero, one or more tuples, based on the mapping condition.
- *Aggregate*, a stateful operator that is used to aggregate information from multiple tuples over a window.

Determinism is a requirement to ensure correct results in spite of varying interleaving of input tuples. Even assuming each input stream delivers its tuples in timestamp-order, it is still challenging to preserve determinism for operators that receive tuples from various streams, as they can be interleaved due to communication. A recently proposed mechanism to support deterministic processing is *ScaleGate* [8], [9], a parallel coordinator that efficiently supports merging of several

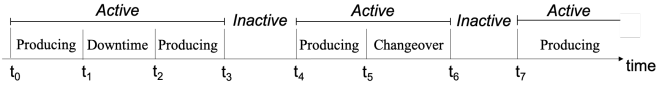


Fig. 2: Active and inactive machine states (adopted from [16])

timestamp-sorted streams, using fine-grained synchronization. It allows for an arbitrary number of reader entities to consume tuples from the merged stream while encapsulating the communication between *sources* and *readers* through defining two methods; (i) `addTuple(tuple,sourceID)`, which allows source entity `sourceID` to add `tuple` to the ScaleGate, and (ii) `getNextReadyTuple(readerID)`, which provides the next tuple in the output stream that has not been consumed yet by the reader entity `readerID`. ScaleGate is a key component of our methods and its use is detailed in the corresponding context in the paper.

III. Problem Overview and Modeling

A. Shifting Bottleneck Detection

In a nutshell, the shifting bottleneck detection method uses the notion *active period* of machines to detect *momentary bottlenecks* of a production system and identifies *sole* and *shifting bottlenecks* over a *selected interval of time* T (e.g. a production run hour, shift, day, week). Further, the method calculates the fractions of T that each machine was a bottleneck and reports the one with the highest fraction as the primary bottleneck.

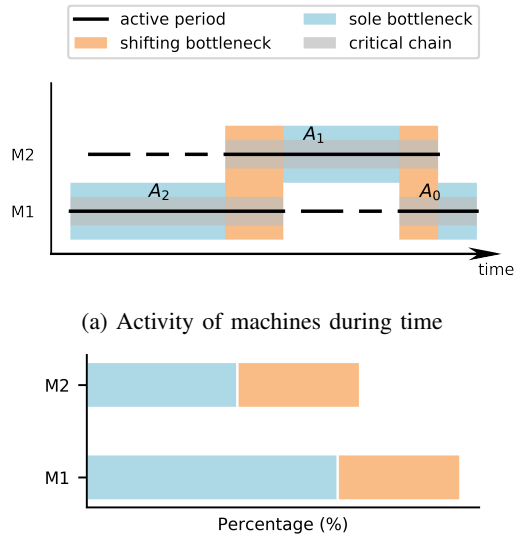
In the following, we introduce terms and refine the descriptions in [17], for ease of reference in subsequent parts of the paper, where we model the problem and describe our proposed stream processing methods.

Definition 1. A machine state is *inactive* when the machine is *waiting* (for e.g. arrival or removal of a part), otherwise it is *active*. An *active period* of a machine is a time interval during which a machine is active without interruption.

To calculate the active periods of a machine, first its states during a production run need to be distinguished into active or inactive (c.f. Figure 2 for an example; notice that consecutive active periods are concatenated). Having this information, the *momentary bottlenecks* are defined as follows.

Definition 2. A *momentary bottleneck* at any time t is the machine i with the longest active period up to t, among all active machines at that moment. Let A^i denote this active period of machine i that causes the latter to be characterised as momentary bottleneck; in the following we will slightly abuse the term and use it for both i and A^i.

The importance of the momentary bottleneck is due to the fact that there is a correlation between machines in a production system, where activity of a machine affects the states of the others, i.e. the longest a machine is active, the highest its influence is on the other machines. Having identified the momentary bottlenecks, the method in [17] continues by defining the *critical chain of momentary bottlenecks* during the selected time interval T.



(a) Activity of machines during time

(b) The percentage of being bottleneck for each machine.

Fig. 3: Illustration of terms in shifting bottleneck method for two machines (adopted from [17])

Definition 3. The *critical chain of momentary bottlenecks* during a time interval T, i.e. (T_{start},T_{end}), is a sequence of active periods, $A^0, ..., A^k$ of different machines, where A^0 is the momentary bottleneck at time T_{end}, A^1 is the momentary bottleneck at time A^0_{start} (i.e. the starting moment of the active period A^0), ..., A^k is the momentary bottleneck at time A^{k-1}_{start}. The chain stops at k because either $A^k_{start} \leq T_{start}$ (i.e. A^k starts before or at the same time as the interval of interest), or there is no active machine at time A^k_{start}.

The critical chain of momentary bottlenecks in T specifies how several machines might affect the performance of each other. It also provides information about the *sole and shifting bottlenecks* during T, as defined in the following:

Definition 4. Given a critical chain of momentary bottlenecks $A^0, ..., A^k$: (i) *shifting bottlenecks* (resp. *sole bottlenecks*) are the machines in the overlap of pairs A_i and A_{i+1} (resp. the non-overlapping parts of each A_i) $i \in \{0, 1, ..., k\}$ in the chain.

Figure 3a illustrates an example of active periods, critical chain of momentary bottlenecks as well as sole and shifting bottlenecks in a production system consisting of two machines.

As the last step in the shifting bottleneck method, the percentages of being sole bottleneck or part of shifting bottlenecks are calculated for all machines in the system. In the end, the one with the highest percentage is defined as the *primary bottleneck* in T, which has the highest impact on the system throughput in the period. Following the same example, Figure 3b shows the percentages of the two machines being sole and shifting bottlenecks. As shown in the figure, since M_1 has the highest percentage, it is defined as the primary bottleneck of the time interval.

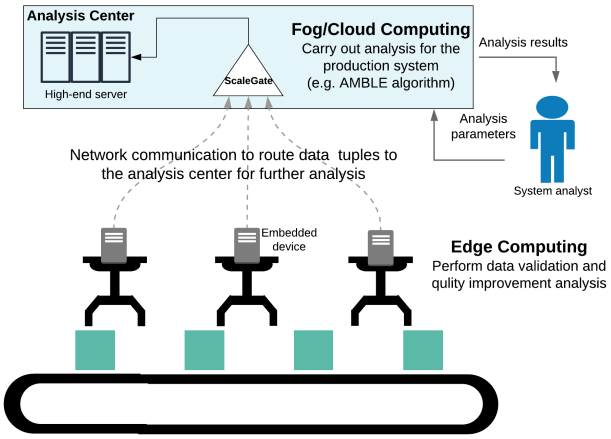


Fig. 4: System model for *STRATUM* framework

B. System and Problem Model

We consider production lines consisting of m machines and an analysis center where data from the machines are gathered, through a communication network, and analysed. We assume the analysis center is deployed at the *fog/cloud* layer, where it employs high-end multi/many-core servers. Furthermore, each machine is connected to MES (i.e. to monitor the machines status) and is equipped with an embedded device. Embedded devices are resource constrained, i.e. provide limited computational power. The analysis that is carried out on these devices is also known as *edge processing*, implying running the analysis close to where data originates.

Based on the system model and the defined notions of the shifting-bottleneck method, we aim at providing (i) a framework to utilize the distributed resources at IoT-based, multi-tier architectures, while improving the quality of data and timeliness of the response as well as reducing contention on the communication system; (ii) an efficient algorithm to process the data while it is being collected, to correctly calculate the momentary, sole, and shifting bottlenecks at any time, with low *latency* (the latter being defined as the timestamp-difference of any result tuple and the latest input tuple that caused it to be produced).

IV. The *STRATUM* Framework

In this section, we present *STRATUM* (STReaming ArchitecTUre for Manufacturing), which is a 2-tier processing framework, combining edge and fog/cloud computing (cf. Figure 4). The lower tier, deployed at the edge, receives a flow of MES data as input and produces a stream of sanitized and organized data for the upper tier. The upper tier, deployed at a fog server or in the cloud, receives the stream of clean and organized data from several sources at the lower tier and carries out the analysis required by the system analyst. The algorithm that we propose for the continuous bottleneck-detection analysis is described in Section V.

A. Data Validation at the Lower Tier

In the edge tier, a flow of MES data as input is received and a flow of sanitized and organized data is produced. Each machine in this tier works independently as an edge node to carry out streaming-based queries. To start composing the queries, the first step is to define the input stream as a sequence of tuples, sharing the same schema. To this end, among all the attributes that a MES data record carries, we keep the creation timestamp, id of the machine, the period during which data is valid, and information about the activity of the machine (in the example system that we use for the experimental evaluation this information is the value of *ANDON*, aka *signal lights*).

Table I shows an example MES record of the production line. As shown in the table, there are four ANDON lights, which are mounted on machines and change the status (i.e. on or off) upon receiving a signal from the machine. At any instance of time, one or several of lights may be on for each machine which indicate the state of machine to be either active or inactive. Table II indicates how combinations of ANDON lights signify a specific state of a machine.

By keeping the above mentioned attributes of MES data record, we use the following schema for input tuples:

$$< ts, machineID, duration, red, yellow, green, white >$$

where ts is the creation timestamp of the tuple, $machineID$ is the id of the production machine, $duration$ shows the validity time interval of the data, and red, $yellow$, $green$, and $white$ are ANDON lights.

timestamp	machineID	duration	ANDON lights			
			r	y	g	w
2017-08-09 14:57:23	M1	1101	1	1	0	0
2017-08-09 15:08:20	M2	190	0	0	1	0
2017-08-09 15:11:30	M2	382	0	1	1	0
2017-08-09 15:15:47	M1	85	1	1	0	0

TABLE I: Example MES record of two machines

ANDON light	Status	State
yellow	producing	active
green		
white		
yellow+white		
red	down	active
no light		
green+yellow	blocked / starved / idle	inactive
green+white		
green+yellow+white		

TABLE II: Combinations of ANDON lights and corresponding machine states

We now define and explain the operators to run the queries on the data stream. Figure 5 shows the proposed graph of streams and operators to be deployed on each machine. The operators that are proposed for this query are standard SPE operators which make the framework usable by engineers with common programming skills. In the following we describe the functionality of each operator.

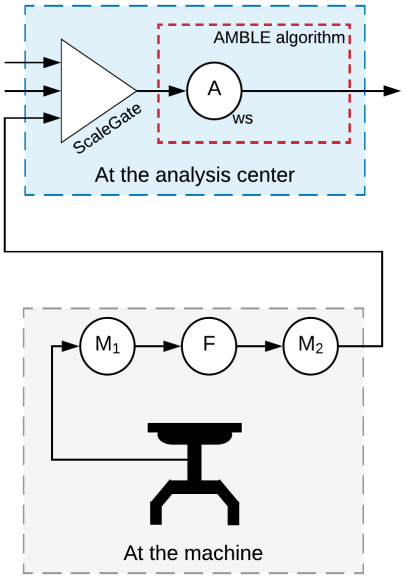


Fig. 5: The directed graph deployed in the *STRATUM* framework to perform the continuous query

- $\texttt{M}_1$ is a Map operator to transform the initial ituple schema into a new one: $< ts, machineID, duration, active >$ where $active$ is a Boolean attribute, computed using Table II and the ANDON lights in the input tuple.
- `F` is a Filter operator to discard all the tuples whose $active$ attribute is `FALSE`. It also discards noise, e.g. the tuples that are outside the scheduled hours. Scheduled hours indicate the time during a day that the production line is active (e.g. from 06:00AM till 11:59PM on Mondays). The output tuples keep the schema of the input ones.
- $\texttt{M}_2$ is another Map operator to organize data, based on the windows defined by the system analyst. Windows show the time granularity that the analyst seeks to identify the shifting bottlenecks through (e.g. every 6 hours). M_2 is used to either forward or split the tuples. If a tuple is located in more than one window, due to its $duration$ attribute, the M_2 operator splits the tuple into two or more with the same schema as before but with updated ts and $duration$.

The output stream of the last operator in the graph is continuously sent to the next tier of *STRATUM*, for continuous shifting bottleneck detection analysis.

B. Data Processing at the Upper Tier

The first step is to merge all tuples from several streams, in a deterministic way in the framework. We employ*ScaleGate*, (Section II), so that sources (the same as machines in this paper) insert a timestamp-sorted stream of tuples each, and readers (processing threads at the analysis center) retrieve tuples in timestamp order from the merged stream. To feed the ScaleGate, each machine runs `addTuple(tuple,sourceID)` method and adds the validated tuples into the ScaleGate using the machine's id as `sourceID`, constantly. The threads operating at the analysis center continuously retrieve the tuples from ScaleGate, using `getNextReadyTuple(readerID)` and perform the analysis.

Concurrent multiple-instance analysis: Notice the ability to have more than one readers in the ScaleGate, allowing several threads to execute the analysis using different input parameters, such as multiple window sizes, to identify bottlenecks during shorter and longer periods, e.g. hourly, diurnal, weekly, monthly, yearly. This ability results to a configurable, multiple-granularity concurrent analysis. It can facilitate to give to the analysts information to support e.g. dynamic pipelines.

In the next section we describe the proposed algorithm to be used as for the analysis at the upper tier.

V. The *AMBLE* Algorithm

The *AMBLE* algorithm (Automated Manufacturing BottLeneck dEtection), as shown in Figure 5, is applied on the input stream as an Aggregate operator, `A`. Here we present the details of *AMBLE* and argue about its correctness and complexity.

A. Algorithm description

The main idea of *AMBLE* is to use a *tree* data structure, whose nodes are active periods of machines; data tuples (nodes) are added to the tree upon reception, in order to maintain all chains of momentary bottlenecks. Those nodes (i.e., active periods) that cannot be momentary bottlenecks are not inserted in the tree at all. *AMBLE* enables fast identification of the critical chain at any time instant, by simply making the chain equivalent to a path in the tree, starting from the node which is the momentary bottleneck at that time.

Algorithm 6 presents the main loop in *AMBLE* (L. 6). The input parameter ws, given by the analyst, is the size of time interval T, during which the analyst is interested to find the bottlenecks, while the set $lastNodes$ is used to keep the latest tuple received from each machine. Once a tuple p is read from ScaleGate, a matching $node$ is created using `CreatNewNode` function (L. 11), with its $parent$ initially set to $null$ and its ts and te fields set to the starting and ending timestamps of the active period of p. (L. 2-5). Once $node$ is created, *AMBLE* checks if the former lies in the current window, by comparing its timestamp with the value of T_{start} and ws (T_{start}, initially set to zero, indicates the starting timestamp of the current window); if not, the algorithm updates the current window, traverses the tree to output the critical chain, and empties the $lastNodes$ set. (L 12-15). Otherwise, if tuple p lies in the current window, the algorithm compares the current node with the ones in the $lastNodes$ set, to find a parent and insert the node in the tree (if needed). The idea is to set the parent-child link in a way that the parent of the node containing tuple p, be the momentary bottleneck at the moment $p.ts$, i.e. the starting moment of the active period of tuple p.

Claim 1. In a system with m machines, the parent of a node p can be found by *AMBLE* through at most m comparisons.

Algorithm 1: Algorithm *AMBLE*

```
1  Function creatNewNode(Tuple p)
2      set parent to null
3      set ts to p.ts        ▷ starting timestamp of the active period
4      set te to p.ts + p.duration        ▷ ending timestamp
5      set machineID to p.machineID
6  Function Main(ws)
7      Node[] lastNodes = ∅
8      firstTimeSeen = true
9      while executing do
10         retrieve p from ScaleGate
11         node = createNewNode(p)
12         while node.ts ≥ T_start + ws do
13             increase T_start by ws        ▷ tumbling window
14             traverse(lastNodes, T_start) (Alg 2)
15             empty lastNodes
16         for l ∈ lastNodes do
17             if l.machineID = node.machineID then
18                 substitute l with node in lastNodes
19                 firstTimeSeen = false
20                 continue
21             if l.te < node.ts then
22                 continue
23             if l.te ≥ node.te then
24                 break
25             if node.parent = null or
                node.parent.ts > l.ts then
26                 node.parent = l
27         if firstTimeSeen then
28             add node to lastNodes
```

Proof. (sketch) Since tuples are processed in starting-timestamp order and there is no overlap between tuples from the same machine, the momentary bottleneck at the starting time of p, $p.ts$, is one of tuples in the set containing the latest tuple seen from each machine. □

Following Claim 1, instead of comparing the specific $node$ (corresponding to p) with all the nodes in the tree, *AMBLE* only looks into the latest node of each machine (L 16), i.e. those in $lastNodes$. While looking for the parent of $node$ among each node l in the $lastNodes$, there are four possibilities: (i) Both $node$ and l originated from the same machine (L. 17), hence it cannot be the parent. *AMBLE* only substitutes l by $node$ in $lastNodes$ set and sets the flag for $firstTimeSeen$ for later on. (ii) The starting time of $node$ is after the ending time of l (L 21). In this case, l can not be the momentary bottleneck at the starting time of $node$, because it is not active at that moment. (iii) The ending time of $node$ is before the ending time of l (L 23). In this case, adding $node$ to the tree is redundant because at any point in time during the $node$'s active period, $node$ can not be a momentary bottleneck. Instead, l, that fully overlaps $node$'s active period and started no later than the latter, can potentially be. Therefore, it is enough to keep l in the tree and not include $node$. (iv) The ending time of l falls between the starting time and the ending time of $node$, which implies l is active at the moment that $node$ is started and therefore, l can be a potential momentary bottleneck prior to $node$. In this case, l is set as the parent of $node$ if either the $parent$ attribute is not set or the $node$'s current parent has a starting time later than l's one (L 26). After the for loop, $node$ is added to the $lastNodes$ set, if it is the first tuple that originated from the corresponding machine in the current window.

Algorithm 2 shows the procedure of traversing the tree at the end of each window, to detect the critical chain, starting from a leaf which is the momentary bottleneck at the end of the window. Following Claim 1, the momentary bottleneck at the ending timestamp of any window is amongst the latest tuples of different machines. In this regard, `momentaryBottleneck` method returns the node from the set $lastNode$ (by iterating on it), which is the momentary bottleneck at timestamp t (Alg. 2 L. 2). Starting from that node, *AMBLE* continues traversing the nodes using the parent-child link until lit reaches the root. The result of traverse is returned as the critical chain.

Algorithm 2: `traverse` function for *AMBLE*

```
1  Function Node[] traverse(lastNodes, t)
2      b = momentaryBottleneck(lastNodes, t)
3      while b.parent ≠ null do
4          add b to results
5          b = b.parent
6      add b to results
7      return results
```

There is a direct match of the steps of the algorithm traversing a directed path in a tree starting from a momentary bottleneck node, with the definition of the critical chain of momentary bottlenecks in Section III. After detecting the critical chain of momentary bottlenecks, the sole and shifting bottlenecks can be found using Definition 4. The above leads to the following claim:

Claim 2. Algorithm *AMBLE*, given sequences of timestamp-sorted tuples describing sequence of machine states (one sequence per machine) over a time interval T, correctly identifies the critical chain of momentary bottlenecks in T and the resulting sole and shifting bottlenecks.

Continuous querying: Notice that at any point in time t (i) the $lastNodes$ set can provide, through an iteration, the momentary bottleneck at t and (ii) traversing the tree from the momentary bottleneck at t up to the root gives the critical chain at that time. With this notice we can deduce that, besides offering the window-based analysis as shown in the previous subsections, the *AMBLE*'s data structures can provide the

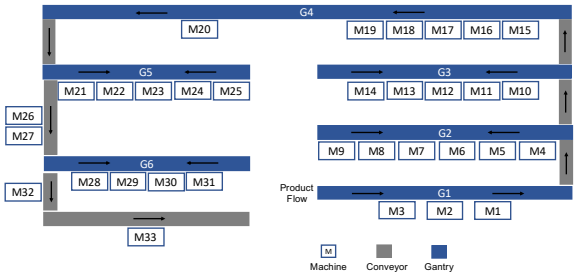


Fig. 6: Layout of the production line

means to support an operation that can request the shifting bottleneck information in a continuous manner, with per input-tuple granularity.

B. Complexity

To study the complexity of *AMBLE*, we distinguish two parts; the complexity of node insertion and the complexity of traversing the tree.

As stated in Claim 1, at most $m-1$ comparisons are needed to insert a node into the tree, where m is the number of machines in the system. Therefore, the complexity of processing a tuple, i.e. a node insertion in the tree depends linearly only on the number of machines, m, implying that for any given system configuration (i.e. given the number of machines) it is constant, independent of the number of tuples.

The complexity of traversing the tree at the end of the period depends on the length of the critical chain of momentary bottlenecks, hence the overhead is a constant per output tuple. In the worst case, this can be at most linear on n (where n is the total number of tuples received from all machines in the specified window size). Such a case is extremely unlikely though, since for this to happen, all tuples from all machines need to be part of the critical chain.

Compared to [20], besides that the analysis in the latter is taking place after all the data is gathered, another significant difference is due to the data structure over which data is processed; in [20], a matrix with m rows and N columns is maintained, N being the number of measurement intervals, typically large, since the measurement intervals are commonly only a few seconds long; the matrix is traversed twice to carry out the calculations, resulting to $O(mN)$ operations.

VI. Experimental Study and Discussion

We evaluate here, the performance of the *STRATUM* framework, running the *AMBLE* algorithm for bottleneck detection, on a dataset taken from a production line in an automotive manufacturing company in Sweden. The production line consists of 33 machines and 6 gantries, where gantries transport the material between the parallel machines, all connected to MES for status monitoring (Figure 6). The dataset, collected over two years, contains approximately 8.7 million records.

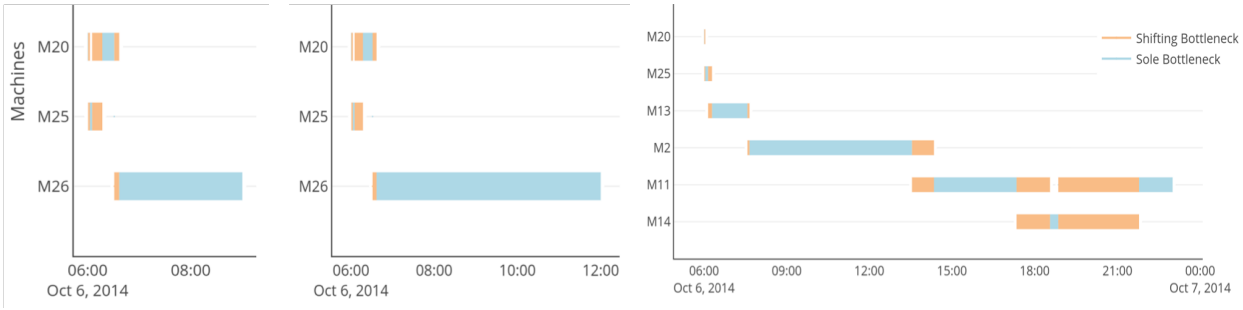
The *STRATUM* framework is implemented in Java on top of the Apache Flink SPE [4]. The *AMBLE* algorithm is also implemented in Java and imported to an operator in the framework, as discussed in Section V. Since inter-tier communication is not in the scope of this work, we run the experiments on a 2.10 GHzIntel(R) Xeon(R) E5-2695 with 38 cores on 2 sockets (18 cores per socket) and 64 GB memory.

As discussed in Section IV, the framework employs a thread to read timestamp-sorted tuples from ScaleGate and perform the analysis over a window of the data stream. Figure 7 shows the result of running the *AMBLE* algorithm on the same data with different window sizes. As shown in the figure, applying different windows results to a different critical chain and accordingly, different pattern of sole and shifting bottlenecks. The reason is that a chain is selected among all others in the *AMBLE* algorithm based on the momentary bottleneck at the end of the window. For instance, at time 12:00 (at the end of the window with ws equal to six hours), Machine $M26$ is the momentary bottleneck. This means, *AMBLE* starts from the leaf containing the last tuple of $M26$ and traverses the tree to detect the critical chain. Nevertheless, by choosing larger window size (e.g. ws is equal to one day) the system gets a wider perspective of the critical chain. In this example, the momentary bottleneck at the end of the one-day window is the latest tuple of machine $M11$, from which *AMBLE* starts traversing the tree. In the new critical chain, $M26$ is not involved.

Since *STRATUM* pipelines collecting and validating data at the lower tier with the analysis at the upper tier, as well as using *AMBLE* to create the internal data structure in streaming fashion, it is expected to get the results in near real-time. Figure 8 shows the average latency of applying *STRATUM* framework including data validation and *AMBLE* algorithm over the data, for different window sizes. As shown in the figure, smaller time granularity causes higher latency because of the time it takes for data validation. By choosing a small window size, many tuples might lie in more than one window, which in turn demand to be split and duplicated. Nonetheless, even with a window size equal to three hours, the average latency is less than one second.

VII. Conclusions and Future Work

We proposed the *STRATUM* framework for data processing in production systems to utilize the distributed resources at multi-tier architectures; *STRATUM* is leveraging data stream processing to improve quality of data and reduce overhead on the communication bandwidth. We also proposed a continuous shifting bottleneck detection algorithm, *AMBLE*, for efficient, automated and configurable continuous processing. In the experimental evaluation using real-world data of 2 years, we show the potential of *STRATUM* using *AMBLE* for high impact on the state of the art. As mentioned in the introduction, the design of algorithmic approaches for stream processing is problem-dependent and hence appropriate methods are required. *AMBLE* is such an example, that is presented here as a streaming operator. In line with approaches in [12]–[14], enabling the algorithmic implementation of *AMBLE* through primitive streaming operators and enhancing the algorithmic implementations with parallelization of the data



(a) ws is equal to three hours (b) ws is equal to six hours (c) ws is equal to one day

Fig. 7: Sole and Shifting pattern of machines at different times during the same day

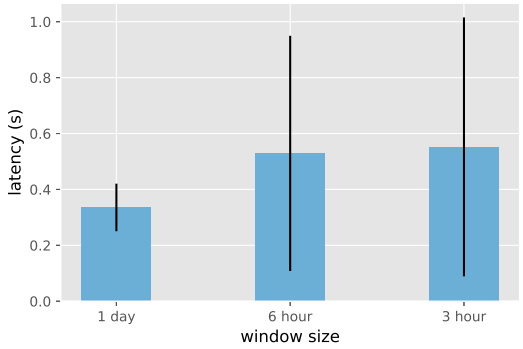


Fig. 8: Average latency of *STRATUM* framework, including data validation at the lower tier and *AMBLE* at the upper level

structures maintenance and with elasticity in dealing with varying streams, are interesting future directions.

Problems related to predictable distributed processing using flexible real-time service-oriented approaches in factory automation, have also been studied in contexts of web-based infrastructures (e.g [5], [10]). Here we study the problems from a continuous processing perspective, working on the flows of data generated in IoT contexts with multiple sensors and show how the multi-tier processing architecture can be utilized for accurate, low-latency processing. A holistic, service-level-agreement related perspective is also an interesting area of study, as also described in [1].

References

[1] M. Ashjaei, K. Clegg, L. Corneo, R. Hawkins, O. Jaradat, V. M. Gulisano, and Y. Nikolakopoulos. Service level agreements for safe and configurable production environments. In *2018 IEEE 23rd Int'l Conf. on Emerging Technologies and Factory Automation (ETFA)*, volume 1, pages 1252–1255. IEEE, 2018.

[2] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16. ACM, 2002.

[3] C. Betterton and S. Silver. Detecting bottlenecks in serial production lines–a focus on interdeparture time variance. *Int'l J. of Production Res.*, 50(15):4158–4174, 2012.

[4] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Comp. Soc. Technical Committee on Data Engineering*, 36(4), 2015.

[5] T. Cucinotta, A. Mancina, G. F. Anastasi, G. Lipari, L. Mangeruca, R. Checcozzo, and F. Rusinà. A real-time service-oriented architecture for industrial automation. *IEEE Trans. on industrial informatics*, 5(3):267–277, 2009.

[6] E. M. Goldratt and J. Cox. *The goal: a process of ongoing improvement*. Routledge, 2016.

[7] M. Gopalakrishnan, A. Skoogh, and C. Laroque. Simulation-based planning of maintenance activities by a shifting priority method. In *Proceedings of the 2014 winter simulation Conf.*, pages 2168–2179. IEEE Press, 2014.

[8] V. Gulisano, Y. Nikolakopoulos, D. Cederman, M. Papatriantafilou, and P. Tsigas. Efficient data streaming multiway aggregation through concurrent algorithmic designs and new abstract data types. *ACM Trans. Parallel Comput.*, 4(2):11:1–11:28, Oct. 2017.

[9] V. Gulisano, Y. Nikolakopoulos, M. Papatriantafilou, and P. Tsigas. Scalejoin: A deterministic, disjoint-parallel and skew-resilient stream join. *IEEE Trans. on Big Data*, 2016.

[10] F. Jammes and H. Smit. Service-oriented paradigms in industrial automation. *IEEE Trans. on industrial informatics*, 1(1):62–70, 2005.

[11] L. Li, Q. Chang, and J. Ni. Data driven bottleneck detection of manufacturing systems. *Int'l J. of Production Res.*, 47(18):5019–5036, 2009.

[12] H. Najdataei, Y. Nikolakopoulos, V. Gulisano, and M. Papatriantafilou. Continuous and parallel lidar point-cloud clustering. In *2018 IEEE 38th Int'l Conf. on Distributed Computing Systems (ICDCS)*, pages 671–684. IEEE, 2018.

[13] H. Najdataei, Y. Nikolakopoulos, M. Papatriantafilou, P. Tsigas, and V. Gulisano. Stretch: Scalable and elastic deterministic streaming analysis with virtual shared-nothing parallelism. In *Proceedings of the 13th ACM Int'l Conf. on Distributed and Event-based Systems*, pages 7–18. ACM, 2019.

[14] D. Palyvos-Giannas, V. Gulisano, and M. Papatriantafilou. Genealog: Fine-grained data streaming provenance at the edge. In *Proceedings of the 19th Int'l Middleware Conf.*, pages 227–238. ACM, 2018.

[15] L. Pehrsson, A. H. Ng, and J. Bernedixen. Automatic identification of constraints and improvement actions in production systems using multi-objective optimization and post-optimality analysis. *J. of manufacturing systems*, 39:24–37, 2016.

[16] C. Roser, M. Nakano, and M. Tanaka. A practical bottleneck detection method. In *Proceedings of the 33nd Conf. on Winter simulation*, pages 949–953. IEEE Comp. Soc., 2001.

[17] C. Roser, M. Nakano, and M. Tanaka. Productivity improvement: shifting bottleneck detection. In *34th Conf. on Winter simulation: exploring new frontiers*, pages 1079–1086, 2002.

[18] M. Stonebraker, U. Çetintemel, and S. Zdonik. The 8 requirements of real-time stream processing. *ACM Sigmod Record*, 34(4):42–47, 2005.

[19] M. Subramaniyan, A. Skoogh, M. Gopalakrishnan, H. Salomonsson, A. Hanna, and D. Lämkull. An algorithm for data-driven shifting bottleneck detection. *Cogent Engineering*, 3(1):1239516, 2016.

[20] M. Subramaniyan, A. Skoogh, H. Salomonsson, P. Bangalore, M. Gopalakrishnan, and A. Sheikh Muhammad. Data-driven algorithm for throughput bottleneck analysis of production systems. *Production & Manufacturing Res.*, 6(1):225–246, 2018.